Interaction Design in Agent-based Service-oriented Computing Systems

José Ghislain QUENUM and **Fuyuki Ishikawa** and **Shinichi Honiden**
National Institute of Informatics
2-1-2 Hitotsubashi, Tokyo 101-8430, Japan

Abstract

This paper presents a new approach which addresses the issue of inconsistent message exchange during agent interactions. We advocate that in agent-based service-oriented computing systems, only agents should be in charge of executing interactions. We also require that the architecture of an agent be clearly separated in two distinct parts, a public and a private parts. The public part contains the interaction model, while any other data and process the agent needs belong to the private part. Our solution consists of automatically constructing the interaction model. It is based on a unification of the actions, required of an agent playing a role in a generic protocol, and the functionalities abstracted from the *BPEL4WS* model of this agent. We present the algorithms to perform this unification as well as the abstract models they manipulate.

Introduction

The recent years have witnessed an increasing interest for *service-oriented computing (SoC)*. SoC consists of modelling, designing and implementing heterogeneous, distributed and open systems embedding several components, which provide and/or request services. In this new paradigm, software components provide services to each other through published and discoverable interfaces, and these services can be invoked over a network.

The Web service research domain has taken a leading part of SoC (Haas 2004). Its related technologies rapidly evolved from basic specifications, e.g., a messaging framework, to standard languages for specifying how to orchestrate existing services in order to compose a new service. *Business Process Execution Language for Web services (BPEL4WS)* is one of these languages. It provides a powerful notation for an executable process interacting with its partners (other components providing or requesting a service) (Andrews *et al.* 2003). *Web Service Choreography Description Language (WS-CDL)* is another standard language, which describes peer-to-peer collaborations of participants by defining, from a global viewpoint, their common and complementary observable behaviour (W3C 2004).

Some research endeavour in multi-agent systems (MAS) scaled the characteristics of SoC systems to the requirements of MAS. In the resulting architecture, agents are supposed to control the components providing and/or requesting services. A special concern, which still remains unsolved, is the execution of interactions during a service provision. In this research, we take a different view on this issue from before. We claim that in such architectures, only agents should be in charge of executing the required interactions so that the services be provided correctly. Behind our position, lies a design guideline which helps identify the relationship between entities in an agent-based SoC, especially between agents and components providing and/or requesting services. More precisely, agents are supposed to bring more "intelligence" and "sociability" in SoC, and interactions are a means to make both features effective.

A common trend in SoC is to represent a concrete protocol (message exchange sequence associated with concrete data types) for each service, and application domain. Additionally, the messages are described following SOAP (Simple Object Access Protocol) specifications, which are only textual description without a precise semantics. In MAS, agent interactions are often based on generic protocols. In these protocols, only a general behaviour of the communicating entities can be provided. Introducing generic protocols in agent-based SoC systems, offers the advantage of using the same protocol for several services and for different application domains. Moreover, most of the formalisms for generic protocol representation describe messages following an agent communication language (ACL). In doing so, they make it possible to provide a meaning for the exchanged messages. This approach is an advantage over SOAP messaging protocol. From what precedes, concrete protocols in SoC can be perceived as a configuration of generic protocols: a refinement (or specialisation) of the general behaviour of the communicating entities. Usually, this configuration is performed by hand, and may introduce inconsistencies in message exchange during service provision.

As an illustration of this issue, consider a transportation MAS, containing among other agents, *trader agents* which request good transportation services provided by *carrier agents*. We also consider that there are two independent designers in the MAS. A designer *A* for *trader agents*, and a designer *B* for *carrier agents*. In our scenario, designer *A* manually configures the initiator role of the Contract Net Protocol (CNP) (Smith 1980) for a trader agent, T_0. Designer *B* does the same with the participant role for an *air carrier agent*, AC_1. The sequence diagram of CNP is given in figure 3. Most generic protocol specifications focus on the description of the messages exchanged during the interaction. They do not propose any pattern for the content of these messages. CNP specifications suffer the same limitation. In our scenario, both designers interpreted the bidding deadline differently. *A* did not consider any deadline for the `call-for-proposals (cfp)` message T_0 will send to the potential service providers. *A* then designed a method, `callForTransportation`, which generates the `cfp` message. The content of the message has the following pattern: *transport good ? from ? to ?*. On the other hand, *B* expects the deadline to be explicitly mentioned in the content of the `cfp`. Therefore, *B* designed a method, `handleCallForTransportation`, to handle the `cfp` message and especially its content. The content pattern this method expects, comes as follows: *transport good ? from ? to ? deadline ?*. At runtime, a `cfp` exchange between T_0 and AC_1 will lead to an inconsistent message exchange. A message exchange is inconsistent, on the receiver's viewpoint, when at least one of the fields in the structure of this message (as defined in the ACL following which this message is represented) or its content does not match the expected pattern. A consequence of this inconsistency is the cancellation of the intended service, the transportation of the good.

To address this issue, we propose an approach to automatically configure generic protocols. Our approach prevents agent designers from deviating from the correct interpretation of a protocol by adopting identical (or at least similar) specifications for this protocol. In the remainder of the paper, we elaborate on the mechanisms our approach is based on.

Automatic Protocol Configuration

The automatic protocol configuration approach we developed unifies the actions executed in the context of a protocol and the internal functionalities of an agent. Unification is a well known technique in artificial intelligence (and several other research areas (Knight 1989)), which consists of checking the similarity between two terms, and if necessary adapt one of them so that the matching becomes effective.

We introduce two new models for an agent: an *interaction model* and a *functional model*. The *interaction model* belongs to the public part of the agent's architecture. It is the final output of the configuration process. It contains a formal description of the sequence of communicative acts an agent commits to with respect to a number of coordination scenarios. These coordination scenarios are inspired from generic protocols, which we assume are specified in a formalism and stored in a public library. Generic protocols specify the external behaviour of agents only in a general way. They do not precisely describe the actions which govern this external behaviour. The configuration mechanism thus helps refine these descriptions, matching them with some elements of the agent's architecture. In a SoC context, we require agent designers to describe the *interaction model* following *WS-CDL* specifications. The behavioural bricks that appear in the *interaction model* are controlled by the *functional model*. This latter belongs to the private part of the architecture. It contains a description of the functionalities of an agent in terms of information transformation during an interaction. It is abstracted from the *BPEL4WS* model of the agent. *BPEL4WS* offers two usage patterns to describe business processes: business protocols (which describe the message exchange between partners) and executable business process (which offers an abstract representation of the business processes implied in a service provision). We agree with (Desai & Singh 2004) that a monolithic representation of business processes has several shortcomings. In our approach we delegate the representation of interactions to generic protocol descriptions. Therefore, we focus on the description of abstract business processes in the *BPEL4WS* model. The abstraction from *BPEL4WS* model to the *functional model* consists of a generalisation of the first. We discuss the rules which govern this generalisation in the following section. We also shed light on the *functional model* representation formalism, which is close to the one we proposed to represent generic protocols. Our modelling approach fulfils our initial claim to separate agents' architecture in two distinct parts. Both parts being related by the *functional model*. These three models (*protocol model*, *functional model* and *interaction model*) play an essential part in our approach. They are manipulated by the algorithms we devised for the automatic configuration.

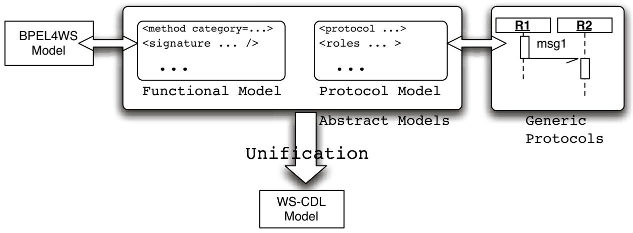


Figure 1: Automatic construction of an agent's *interaction model*

Figure 1 depicts our configuration approach. At the heart of the mechanism are the *functional model* and a standard library of generic interaction protocols, represented in an abstract model. Given a library of generic protocols, our automatic configuration process starts with the design of the functional model. Next, for each role of a protocol the designer wishes to support, we look for similarities between the functionalities (in the functional model) and the actions which are executed by this role. This similarity checking is achieved by a *matching algorithm*. The algorithm identifies the possible similarities between the agent's methods and the actions required by the protocol. When every action in a role (any communicating entity in a protocol) is not paired with a method, an *adaptation algorithm* immediately provides the designer with the specifications of the missing functionalities to support the protocol (actually the role) at hand. Finally, when a role is completely unified (all the actions in the definition of the role are associated with functionalities), we explicitly link each action and the corresponding method in the *WS-CDL* model associated with the configured role. Our mechanism is organised in four steps:

1. abstract representation of generic interaction protocols following a formalism we developed;
2. abstract representation of the functionalities the agent will use during interactions;
3. matching of actions of generic protocols (actually roles) to the functionalities of the agent;
4. insertion of the configured roles into the interaction model, following *WS-CDL* specifications.

Abstract Models

Protocol Model

Several formalisms have been proposed to represent agent interaction protocols, e.g., (Casella & Mascardi 2001),(Domingue, Galizia, & Cabral 2006) (Walton 2005). However, neither these formalisms emphasise the representation of actions needed to generate or handle the exchanged messages, nor do they fulfil our claim to separate interaction concerns from agents' internal activities. To address this weakness, we developed a generic protocol representation formalism, which is based on the AUML protocol diagram (Bauer & Odell 2005). Thereof, we identified five fundamental concepts: *protocol*, *role*, *event*, *action* and *phase*. We formally define these concepts in the remainder of the section. The formal specifications we developed to represent generic protocols using these concepts are discussed in (Quenum *et al.* 2006). The description of these specifications is out of the scope of this paper.

Definition 1. Generic Protocol
A generic protocol is a collection of roles R, which interact with one another through message exchange. The messages belong to a collection M, and are described following an ACL e.g., FIPA ACL (FIPA 2001), KQML (Labrou & Finin 1997). The exchange sequence is represented by Ω. A protocol also has some intrinsic properties Θ (attributes and keywords) which are propositional contents that provide a context for further interpretation of the protocol. We note $P \stackrel{def}{=} < \Theta, R, M, \Omega >$.

Definition 2. Role *A role consists of a collection of phases, Π (see definition 4). It can have some global actions, A_g, which are executed outside all phases and some data other than message content,* variables V. $\forall r \in R, r \stackrel{def}{=} < \Theta_r, \Pi, A_g, V >$, *where Θ_r corresponds to the role's intrinsic properties (e.g., cardinality) which are propositional contents that help further interpret the role.*

The behaviour of a role is governed by events. An event is perceived as an atomic change which occurs in the environment of the MAS. We consider several types of events: message reception, variable value change, message generation, etc. Actions help describe the behaviour of a role.

Definition 3. Action
An action is an operation a role performs while executing. This operation transforms the whole environment or the internal state of the agent currently playing this role. An action has a category ν, a signature Σ and a set E of events it reacts to or produces. We note $a \stackrel{def}{=} < \nu, \Sigma, E >$.

Since the protocols we specify are generic, it is not possible to provide a concrete and complete description of the actions required by these protocols. We fix this limitation by introducing action categories, so that we can provide a semantics for actions in generic protocols. These categories also serve as a classification means. We consider six action categories. (1) *append*, to append a data to a collection; (2) *remove*, to remove a data from a collection; (3) *set*, to assign a value to a data; (4) *update*, to update the value of a variable (5) *send*, to send some information to another agent; and (6) *compute*, to perform a (more complex) computation.

Definition 4. Phase
Some successive actions sharing direct links can be grouped together. Each group is called a phase. Two actions share a direct link if the input arguments (or only a part of them) of one are generated by the other (f.i. when a send *action sends the message generated in a prior action). We note $\forall p_h \in \Pi, p_h \stackrel{def}{=} < A, \prec >$, where A is a set of actions and $\prec$ a causality relation.*

The concept of phase helps define the behaviour of a role through smaller logical blocks.

Agent Functional Model

The functional model contains the set of functionalities an agent can achieve during collaborative tasks. Each functionality is called a *method*[1]. We note $f \stackrel{\text{def}}{=} \{m_\imath\}$. A method is represented by: (1) its category ν, the same as in actions of protocols; (2) its signature Σ, where we introduce user-defined data types in addition to the built-in types used in action signature; (3) its constraints C, immediately following methods (succession constraints) and those excluded once the current method has completed (exclusion constraints). We note $m_\imath \stackrel{\text{def}}{=} < \nu, \Sigma, C >$.

The functional model is a generalisation of the *BPEL4WS* model. Actually the protocol representation formalism does not comply with the *BPEL4WS* specifications (f.i., activity and action though similar concepts cannot be automatically compared because of their initial descriptions). However, our approach needs to match elements in both models and none of the formalisms can be modified. Indeed, *BPEL4WS* is a well established standard in SoC. And to our knowledge, our protocol representation formalism is the one, thus far, to allow the description of generic protocols needed in open and heterogeneous MAS. Hence, we generalise the *BPEL4WS* into a formalism, which is compliant with the protocol representation formalism. Note that we developed some formal specifications for the functional model, which we do not discuss in this paper. In the remainder of this section, we present the rules for the generalisation. For this purpose, we introduce a new relation `generalise`, which we define as follows.

[1]The term method, here, has nothing to do with object-oriented programming.

Definition 5. *Given a business process b_p and a functional model f, f generalises b_p (notation being* $\texttt{generalise}(f, b_p)$*) iff* $\forall act \in b_p, \exists m \in f$

- *given the semantics of any activity act of the business process, it can be classified in the category m belongs to;*
- *the data types act handles or generates are compatible with the signature of m;*
- *given the position of act in the process flow, the constraints placed on m comply with the links between act and its neighbours.*

Property 1. *Given a business process, b_p, there exists a collection of methods $\{m_i\}$ which generalises b_p.*

Proof. We discuss the proof of this property in three stages.

category in our protocol representation formalism, we consider six action categories: *append*, *remove*, *set*, *update*, *send*, and *compute*. The semantics of an activity in a business process always permits its classification in one of these categories.

signature we copy the data types of the variables each activity manipulates as input arguments and output result. These data-types, when they are not equivalent to the built-in types used for action signature in our protocol representation formalism, should inherit from these built-in types.

constraints the syntactic constructs (sequence, flow, etc.) available in *BPEL4WS* make it possible to derive the succession and exclusion constraints placed on methods.

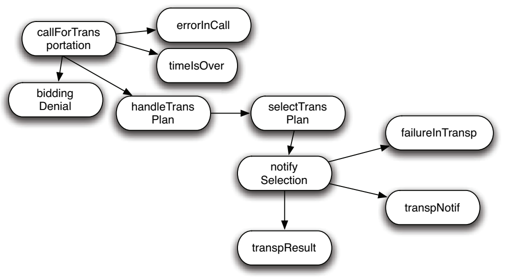
□

Using these rules, we can consider the whole *BPEL4WS* model as a graph, and generalise each node of this graph. By doing so, we obtain a graph similar to the initial one, which corresponds to the agent's functional model. Our generalisation rules thus guarantee that the semantics of the functional model is equivalent to that of the *BPEL4WS* model.

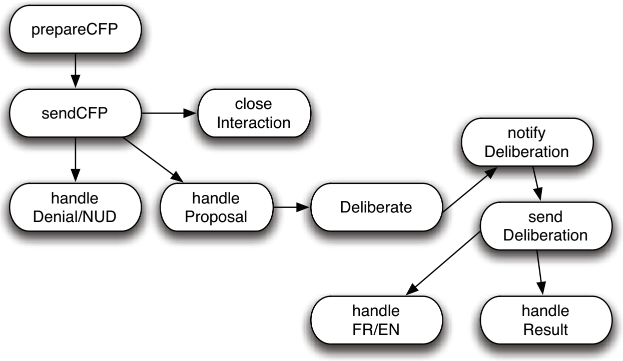
Unification

Algorithms

The unification process consists of a matching mechanism possibly followed by an adaptation mechanism. Algorithm 1 achieves the matching part of the unification. As described above, the process starts with two models: a *functional model* and a *protocol model*. The methods in the functional model are represented as nodes of a directed graph, G_m. The edges in this graph correspond to succession constraints placed on methods: an edge $m_i m_j$ means that m_j can be executed once m_i. Note that the graph corresponding to a functional model may not be connected. As an example, Figure 2(a) contains a partial representation of the graph corresponding to T_0's functional model. For each role the designer wants to support, its actions are represented as nodes of a directed graph, G_a. The edges in this graph express the causal links between actions, derived from the events. Figure 2(b) depicts the action graph of the initiator role of CNP. Our matching process is actually a graph unification. But



(a) T_0's partial method graph



(b) Action graph of the initiator role in the CNP

Figure 2: Example of action and method graphs

we dramatically reduced its complexity by exploring only one graph, the action graph. Adopting a breadth-first exploration, we compare each node of G_a to all the methods in G_m at the same level, considering successful prior matchings. The comparison takes place in three stages: (1) category: check whether the method and the action being compared have the same category; (2) signature: check whether the signature of the method is a valid instance of that of the action; (3) constraints: check whether the constraints of the method comply with the events associated with the action. When there are several methods an action may correspond to, we consider this situation as a conflict. As soon as a successor of this action (an adjacent node) is paired with a method, we try, if possible, to resolve existing conflicts. Properties 2 and 3 discuss the termination and the complexity of this algorithm respectively.

Property 2. *Given a collection R of roles to configure, and an agent functional model, algorithm 1 always terminates.*

Proof. The proof of the termination lies behind that of the non existence of an infinite loop. Indeed, the matching process is taken for a finite set of roles. And every action, from the finite set of actions associated with each role, is explored once. Additionally, each action can be compared only to a finite number of methods in the functional model. Consequently, algorithm 1 always terminates.

□

Algorithm 1 Matching Action-Method

```
Input: Set $Methods$ (methods of the agent); Set $Roles$;
Result: Associative-Array $Unified$; Set $Unpaired$;
  $G_m \leftarrow DiGraph(Methods)$;
  for all role in $Roles$ do
    initialise $explorer$ and $candidates$ to $\emptyset$;
    $actions \leftarrow genActionSet(role)$;
    $G_a \leftarrow DiGraph(actions)$;
    insert initial($G_a$) into $explorer$;
    map initial($G_a$) onto initials($G_m$) in $candidates$;
    while $explorer$ is not empty do
      $curact \leftarrow first(\texttt{explorer})$;
      $result \leftarrow match(curact, \texttt{candidates}.get(curact))$;
      if result is empty then
        insert curact into $Unpaired$;
      else if result is a singleton $\{m_k\}$ then
        for all act adjacent node of curact do
          if act is not explored yet then
            insert act into $explorer$;
          end if
          map act onto $m_k$'s children in candidates;
          map curact onto $m_k$ in $Unified$; resolve awaiting conflicts;
        end for
      else
        for all act child node of curact do
          if act is not explored yet then
            insert act into $explorer$;
          end if
          map curact onto result in $conflicts$;
          map in $candidates$ act onto each child node of each result's element;
        end for
      end if
    end while
    insert all the keys into $conflicts$ and $unpaired$;
  end for
```

Property 3. *For each role to configure, represented by an action graph G_a with v action nodes and e connections between these nodes, algorithm 1 completes in time proportional to $v + e$. As a consequence, given a collection of n roles, each represented by an action graph, algorithm 1 completes in time proportional to $n(v_{max} + e_{max})$, where v_{max} and e_{max} are the node and edge counts respectively, in the largest graph.*

Proof. The complexity of a Breadth-First exploration of a graph (with v nodes and e edges) using adjacency lists is known to complete in time proportional to $v + e$. Since we configure several roles during the process, the complexity evaluates (at most) to a value proportional to n times the sum of the number of the nodes and edges of the largest graph (role). Moreover, each action in G_a is compared to methods in G_m whose adjacent nodes matched the adjacent nodes of the current action. Let φ be the cost of this comparison. For each method, the comparison is held for the category, the signature and the events/constraints. Comparing categories is the simplest of the three comparisons; it consists of strings equivalence. The signature correspondence is generally of low cost by introducing some pattern matching techniques with regular expressions. Finally, events versus constraints comparison demands some inference in order to deduce the causality links as well as the exclusion between actions. Roughly, these comparisons prove not to be costly, since the number of methods is not greater than the number of nodes in any of the connected sub-graph. Once a comparison is held, the decision to make consists of storing some information whether in a collection or in an associative array, which is also of low cost. As a conclusion, φ is of low cost.

$\square$

For actions which do not have any correspondent in the functional model as well as actions involved in conflicts that the algorithm did not resolve, we propose an adaptation. This is achieved by algorithm 2, which generates the description of each unpaired action in the form of a method.

Algorithm 2 Action Adaptation

```
Input: S, set of unpaired actions
Result: S', set of generated methods
  for all action $a_i \in$ S do
    initialise $m_i$;
    set $a_i$'s category to $m_i$;
    derive $m_i$'s signature from that of $a_i$;
    identify the constraints to place on $m_i$;
    insert $m_i$ into S';
  end for
```

Illustration

As an illustration of our approach, we discuss the matching between `prepareCFP` and `callForTransportation`. `prepareCFP` is the initial action of the initiator role of CNP, whose sequence diagram is given in figure 3. `callForTransportation` is one of the initial methods in T_0's functional model. The action graph and method graph both the action and the method belong to are depicted in figure 2(a) and figure 2(b), respectively.

To let the reader have a better understanding of the matching, we sketch out the specifications for CNP and the functional model following the formalisms we developed purposively. Actually, the specifications are written in XML. But here, we adopt a bracket-based representation for the sake of readability. Note that these specifications are a simpler and incomplete version of the ones we developed in (Quenum *et al.* 2006).

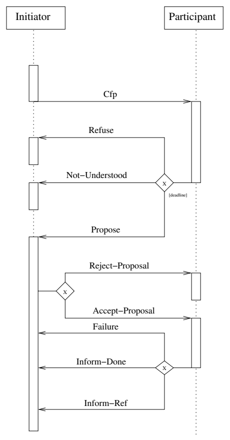


Figure 3: The Contract Net Protocol

```
(protocol (properties ...)
 (roles (role id='initiator'
  (variables  ...)
 (actions ...)
  (phases (phase id='phs1'
 (actions (action category='compute' desc='prepareCFP'
   (signature(arg type='date' dir='in')
   (arg type='any' dir='out'))
   (events (event type='variablecontent' dir='in'
     object='deadline')
    (event type='messagecontent' dir='out'
      object='cfp' id='evt2')))
  (action category='send' description='sendCFP'
   (signature (message id='cfp'))
   (events(eventref dir='in' id='evt2')
   (eventref type='custom' dir='out' id='cus01')
    (event type='endphase' dir='out' id='evt3'))))))))
 (messagepatterns ...))
```

```
(agent
 (types
  (bits (bit id='boolean' name='Boolean')
        (bit id='string' name='String')...)
  (udts(udt id='transdesc' name='TransportDescriptor'
    supertype='any')...))
 (methods(method id='m0' (name='callForTransportation'
   category='compute')
    (signature (arg typeid='date' dir='in')
      (arg typeid='cfp' dir='out')))
  (method id='m3' (name='storeTransportationProposal'
    category='append')
  (signature (arg typeid='proposal' dir='in')
    (arg typeid='proposals' dir='inout'))
  (constraints (constraint status='executed' id='m0')
   (constraint status='notexecuted' id='m1')))))
```

In these specifications, the reserved word *eventref* used in the CNP specifications serves for causality between actions. In addition, the reserved words *bits* and *udts* define built-in types and user-defined types respectively. The matching between `prepareCFP` and `callForTransportation` is detailed in the few lines below. As stated above, the matching process takes place in three stages:

1. category: both have the same category, which is *compute*;
2. signature: From our specifications, it turns out that the signature of `prepareCFP` can be instantiated as that of `callForTransportation`. Indeed, the input argument of `prepareCFP` is a `date` (a built-in type) while the output result is an object of type `any` (a built-in type too). On the other hand, `callForTransportation` handles a `date` data and generates a data of type `cfp`. Since `cfp` is a user-defined type, inherited from `any`, the signature of `callForTransportation` is a valid instance of that of `prepareCFP`.
3. events versus constraints: On the one hand, `callForTransportation` is one of the initial methods of the graph. Therefore, no constraints (follow and exclusion) is associated with it. On the other hand, from the events associated with `prepareCFP`, the corresponding constraint set is empty. As a conclusion, the events in `prepareCFP` comply with the constraints on `callForTransportation`.

Translating Configured Roles

When every action of a role is paired with a method, we consider this role as configured. As a final step in our approach, we generate the specifications for configured roles. We offer two representations of a configured role: (1) an abstract representation following WS-CDL, and (2) a concrete representation in the form of a finite state automaton (FSA), similarly to the approach in (Romero-Hernandez & Koning 2004). Note that in cases where several activities are to be launched at the same time (f.i., the flow construct in *BPEL4WS*), a FSA might not be suitable. In such cases, designers can resort to formalisms like Petri-Nets.

The automaton we generate contains three types of states: *initial*, *final* and *standard* states. As well, four types of events are considered in our translation. (1) *message reception*, (2) *timeout*: a delay has elapsed and an action should be taken, (3) *done*: an internal operation has been executed and generated the expected result, (4) *true*: which are associated with automatic transitions. The translation converts actions into transitions. The causality between actions help identify the correct transition sequence. Thus, actions without predecessors correspond to transition from an initial state. As well, actions without successors correspond to transitions to a final state. Actions having a predecessor and/or a successor correspond to transition from and/or to a standard state.

Property 4. *An action a_k corresponds to a transition $t_{\imath\jmath}$ from state $S_\imath$ to state $S_\jmath$* iff

source state $S_\imath$:

- *a_k has no predecessor and $S_\imath$ is an initial state, or;*
- *a_k follows another action a_{k-1} which transitions to $S_\imath$;*

destination state $S_\jmath$:

- *a_k precedes an action a_{k+1} which transitions from $S_\jmath$, or;*
- *a_k has no successor and $S_\jmath$ is a final state.*

Additionally, we generate an abstract model for further reasoning about agent interactions. This model is the agent's interaction model, and is represented following *WS-CDL* specifications. One advantage we benefit from our protocol representation formalism, is its closeness to *WS-CDL* specifications: most of the specifications constructs can be easily found in our formalism. For instance, *roleType*, *relationshipType* and *participantType* can be extracted from the general description of protocols. Moreover, *informationType* corresponds to all the variable and message content types used in the protocol and enriched with the user-defined types from the functional model. In this paper, we do not formally prove the correspondence. Rather, we emphasise some clues to show evidence of this correspondence. Table 1 defines the correspondence between the concepts in our protocol representation and those in *WS-CDL* specifications. However, some concepts like *channels* and *semantics*

WS-CDL	*Our Formalism*
roleType	role properties
relationshipType	protocol properties
participantType	protocol and role properties
informationType	variable and message content
choreography	exchange sequence
channelType	undefined
workunit	properties and events
activities	actions
interaction activity	send actions
semantics	external

Table 1: Correspondence between *WS-CDL* and our Formalism

are to be explicitly added to our protocol specifications in order to fully comply with *WS-CDL*.

Using the correspondence, we automatically translate the configured role from our formalism to *WS-CDL*. During our translation, we allocate the required channels to establish communication between agents during interactions. As far as the semantics is concerned, we define an external file corresponding to the semantics of each role, which we introduce in the *WS-CDL* specifications.

Related Work

Many studies in SoC focused on the abstraction of public and private parts of agent (and service) architecture, for verification purpose. Some of these studies ((Domingue, Galizia, & Cabral 2006; Desai & Singh 2004; Baldoni *et al.* 2005; Foster 2006) to quote a few) addressed issues relevant to the relationship between the specifications of protocols and the behaviour of each of the partakers (requesters and providers) to a service provision. Especially, most of these studies (f.i., (Baldoni *et al.* 2005; Foster 2006)) aim at checking the conformance of each partaker's behaviour to an adopted protocol, under the assumption that formal descriptions of concrete protocols (configured version of generic protocols) are given. Although our matching and adaptation algorithms can be used for this purpose as well, our work originally supports situations where concrete protocols are not given formally, as discussed below.

The from scratch design of interaction protocols that cover normal as well as faulty situations, is a tedious activity and inconsistency prone. Generic protocols have been introduced in MAS research to allow for reuse of typical interaction patterns for task allocation, auctions, etc. Interaction patterns for Web services often refer to messaging patterns, e.g., send and then receive, and receive multiple messages until end-notification, see (Barros & Boerger 2005). Generic protocols combine such messaging patterns and provide conversation patterns based on goals of interaction. Several studies (f.i., (Paurobally. & Jennings 2005)) attempt to introduce generic protocols into existing SoC technologies. They focused on the support of ACL performatives by means of translator gateways or extension of existing standards for Web Services. However, relationships between abstract generic protocols and concrete protocols, or choreography, have not been discussed so far. Our approach enables to configure role specifications of generic protocols and obtain those of concrete protocols. This approach allows programmers to develop concrete specifications of choreography while making use of generic protocols to implement interaction.

Conclusion

In this paper we discuss some architectural aspects of SoC, where agents play a central role by controlling the components which provide services or request them. Mainly, we focus on the design of the interaction model of such agents by configuring roles of generic protocols. In our approach, the configuration process is automated. Doing so, agent interactions can be executed consistently with respect to initial generic protocol specifications. Clearly, the inconsistent message exchange situation we faced in the introductory example, and which is due to the non compliant representations of `callForTransportation` and `handleCallForTransportation` is fixed using our approach.

As another advantage of our approach, a designer who wishes to support the same protocols through several executions can dramatically reduce the modelling step. He (she) can develop the new functional model by taking inspiration from both the functional and the interaction models. Accordingly, the generation of the new functional model goes faster and in this case there will be no need for an adaptation of this functional model. As well, by separating the interaction and functional models, there can be flexible updates any of them.

This approach, in a more general version, has been implemented (in Java and XML) and applied to two non-trivial application projects, namely Safir (www.projet-safir.org) and Princip (www.princip.net), which both use multi-agent systems to realise sophisticated information retrieval systems. The applicability of the method has then been proved on several tens of generic interaction protocols. Likewise, we envision to apply the approach to real world agent-based SoC systems.

As a future work, we wish to provide agents with the ability to dynamically configure their interaction model. In fact, an agent might need, at runtime, to execute a protocol (as initiator or participant) that does not exist in its interaction model yet. Then, the approach we described in this paper should be automatically and dynamically applied. For this purpose, the agent (the algorithm it uses) is requested to know about the connection between the goals to achieve or task to execute, and the generic versions of interaction protocols the system is provided with, in order to select a set of generic interaction protocols that could be fit.

Furthermore, since agents may have many protocols (precisely roles of these protocols) at their disposal, the issue of protocol selection is raised. This issue is usually solved in a rather simple way. We think that since agents dispose of an explicit interaction model, they might be able to dynamically select a protocol whether to respond to a received message or to achieve a goal (start a protocol). This dynamic protocol selection better suits open and heterogeneous systems, and SoC systems are endowed with both features.

References

Andrews, T.; Curbera, F.; Dholakia, H.; Goland, Y.; Leymann, F.; Liu, K.; Roller, D.; Smith, D.; Thatte, S.; Trickovic, I.; and Weewarana, S. 2003. Business Process Execution Language for Web Services, version 1.1. http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/.

Baldoni, M.; Baroglio, C.; M., A.; Patti, V.; and Schifanella, C. 2005. Verifying the Conformance of Web Services to Global Interaction Protocols: A First Step. In *International Workshop on Web Services and Formal Methods (WS-FM 2005)*, 257–271.

Barros, A., and Boerger, E. 2005. A Compositional Framework for Service Interaction Patterns and Interaction Flows. In *The Seventh International Conference on Formal Engineering Methods (ICFEM'2005)*, 5–35.

Bauer, B., and Odell, J. 2005. UML 2.0 and Agents: how to Build Agent-based Systems with the new UML Standard. *Journal of Engineering Applications of Artificial Intelligence* 18:141–157.

Casella, G., and Mascardi, V. 2001. From AUML to WS-BPEL. Technical report, Computer Science Department, University of Genova, Italy.

Desai, N., and Singh, M. 2004. Protocol-based Business Process Modelling and Enactment. In *The IEEE International Conference on Web Services (ICWS)*, 35–42.

Domingue, J.; Galizia, S.; and Cabral, L. 2006. The Choreography Model for IRC-III. In *Proceedings on Hawai International Conference for System Sciences*, 62–70.

FIPA. 2001. Fipa Communicative Act Library Specification. Technical report, Foundation for Intelligent Physical Agents.

Foster, H. 2006. LTSA - WS-Engineer Eclipse Plugin. http://www.doc.ic.ac.uk/ltsa/bpel4ws/.

Haas, H. 2004. Web Services. http://www.w3.org/2002/ws/. (Access: May 2005).

Knight, K. 1989. Unification: A Multidisciplinary Survey. ACM Computing Surveys.

Labrou, Y., and Finin, T. 1997. A proposal for a new KQML Specification. Technical report, University of Maryland Baltimore County (UMBC).

Paurobally., S., and Jennings, N. R. 2005. Protocol Engineering for Web Services Conversations. *Int. Journal of Engineering Applications of Artificial Intelligence* 18(2):237–254.

Quenum, J. G.; Aknine, S.; Briot, J.-P.; and Honiden, S. 2006. A Modelling Framework for Generic Agent Interaction Protocols. In Endriss, U., and Boldoni, M., eds., *Proceedings of the Fourth International Workshop on Declarative Agent Languages and Technologies*.

Romero-Hernandez, I., and Koning, J.-L. 2004. State Controlled Execution for Agent-object Hybrid Languages. In *Proceedings of the International School and Symposium on Advanced Distributed Systems (ISSAD)*, 78–90.

Smith, G. 1980. The contract Net Protocol: High-level Communication and Control in a Distributed Problem Solver. *IEEE Trans. on Computers* 29(12):1104–1113.

W3C. 2004. Web Services Choreography Description Language version 1.0. http://www.w3.org/TR/2004/WD-ws-cdl-10-20040427/.

Walton, C. 2005. Protocols for Web Service Invocation. In *Proceedings of the AAAI Fall Sympossium on Agents and Semantic Web (ASW05)*.